Proxy-Reflect vue2-vue3响应式原理

王红元 coderwhy

监听对象的操作

- **我们先来看一个需求：有一个对象，我们希望监听这个对象中的属性被设置或获取的过程**
  - 通过我们前面所学的知识，能不能做到这一点呢？
  - 其实是可以的，我们可以通过之前的属性描述符中的存储属性描述符来做到；
- 左边这段代码就利用了前面讲过的 Object.defineProperty 的存储属性描述符来对属性的操作进行监听。
- **但是这样做有什么缺点呢？**
  - 首先，Object.defineProperty设计的初衷，不是为了去监听截止一个对象中所有的属性的。
    - ✓ 我们在定义某些属性的时候，初衷其实是定义普通的属性，但是后面我们强行将它变成了数据属性描述符。
  - 其次，如果我们想监听更加丰富的操作，比如新增属性、删除属性，那么Object.defineProperty是无能为力的。
  - 所以我们要知道，存储数据描述符设计的初衷并不是为了去监听一个完整的对象。

```
Object.keys(obj).forEach(key => {
  let value = obj[key]
  Object.defineProperty(obj, key, {
    set: function(newValue) {
      console.log(`监听到给${key}设置值`)
      value = newValue
    },
    get: function() {
      console.log(`监听到获取${key}的值`)
      return value
    }
  })
})
```

Proxy基本使用

- 在ES6中，新增了一个**Proxy类**，这个类从名字就可以看出来，是用于帮助我们创建一个**代理**的：
  - 也就是说，如果我们希望监听一个对象的相关操作，那么我们可以先创建一个代理对象（Proxy对象）；
  - 之后对该对象的所有操作，都通过代理对象来完成，代理对象可以监听我们想要对原对象进行哪些操作；
- 我们可以将上面的案例~~用Proxy来实现~~一次：
  - 首先，我们需要new Proxy对象，并且传入需要侦听的对象以及一个处理对象，可以称之为handler；
    - ✓ const p = new Proxy(target, handler)
  - 其次，我们之后的操作都是直接对Proxy的操作，而不是原有的对象，因为我们需要在handler里面进行侦听；

```
const obj = {
  name: "why",
  age: 18
}

const objProxy = new Proxy(obj, {})
```

Proxy的set和get捕获器

- 如果我们想要侦听某些具体的操作，那么就可以在handler中添加对应的捕捉器（Trap）：
- set和get分别对应的是函数类型；
  - set函数有四个参数：
    - ✓ target：目标对象（侦听的对象）；
    - ✓ property：将被设置的属性key；
    - ✓ value：新属性值；
    - ✓ receiver：调用的代理对象；
  - get函数有三个参数：
    - ✓ target：目标对象（侦听的对象）；
    - ✓ property：被获取的属性key；
    - ✓ receiver：调用的代理对象；

```
const objProxy = new Proxy(obj, {
  has: function(target, key) {
    console.log("has捕捉器", key)
    return key in target
  },
  set: function(target, key, value) {
    console.log("set捕捉器", key)
    target[key] = value
  },
  get: function(target, key) {
    console.log("get捕捉器", key)
    return target[key]
  },
  deleteProperty: function(target, key) {
    console.log("delete捕捉器")
    delete target[key]
  }
})
```

Proxy所有捕获器

- **13个活捉器分别是做什么的呢？**
- handler.getPrototypeOf()
  - Object.getPrototypeOf 方法的捕捉器。
- handler.setPrototypeOf()
  - Object.setPrototypeOf 方法的捕捉器。
- handler.isExtensible()
  - Object.isExtensible 方法的捕捉器。
- handler.preventExtensions()
  - Object.preventExtensions 方法的捕捉器。
- handler.getOwnPropertyDescriptor()
  - Object.getOwnPropertyDescriptor 方法的捕捉器。
- handler.defineProperty()
  - Object.defineProperty 方法的捕捉器。
- handler.ownKeys()
  - Object.getOwnPropertyNames 方法和 Object.getOwnPropertySymbols 方法的捕捉器。
- **handler.has()**
  - in 操作符的捕捉器。
- **handler.get()**
  - 属性读取操作的捕捉器。
- **handler.set()**
  - 属性设置操作的捕捉器。
- **handler.deleteProperty()**
  - delete 操作符的捕捉器。
- handler.apply()
  - 函数调用操作的捕捉器。
- handler.construct()
  - new 操作符的捕捉器。

Proxy的construct和apply

- 当然，我们还会看到捕捉器中还有construct和apply，它们是应用于函数对象的：

```
function foo() {
  console.log("foo函数被调用了", this, arguments)
  return "foo"
}

const fooProxy = new Proxy(foo, {
  apply: function(target, thisArg, otherArgs) {
    console.log("函数的apply侦听")
    return target.apply(thisArg, otherArgs)
  },
  construct(target, argArray, newTarget) {
    console.log(target, argArray, newTarget)
    return new target()
  }
})
```

Reflect的作用

- Reflect也是ES6新增的一个API，它是**一个对象**，字面的意思是**反射**。
- **那么这个Reflect有什么用呢？**
  - 它主要提供了很多操作JavaScript对象的方法，有点像Object中操作对象的方法；
  - 比如Reflect.getPrototypeOf(target)类似于 Object.getPrototypeOf()；
  - 比如Reflect.defineProperty(target, propertyKey, attributes)类似于Object.defineProperty() ；
- 如果我们有Object可以做这些操作，那么**为什么还需要有Reflect这样的新增对象**呢？
  - 这是因为在早期的ECMA规范中没有考虑到这种对 **对象本身** 的操作如何设计会更加规范，所以将这些API放到了Object上面；
  - 但是Object作为一个构造函数，这些操作实际上放到它身上并不合适；
  - 另外还包含一些类似于 in、delete操作符，让JS看起来是会有一些奇怪的；
  - 所以在ES6中新增了Reflect，让我们这些操作都集中到了Reflect对象上；
- 那么Object和Reflect对象之间的API关系，可以参考MDN文档：
  - https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Reflect/Comparing_Reflect_and_Object_methods

Reflect的常见方法

- Reflect中有哪些常见的方法呢？它和Proxy是一一对应的，也是13个：
- Reflect.getPrototypeOf(target)
  - 类似于 Object.getPrototypeOf()。
- Reflect.setPrototypeOf(target, prototype)
  - 设置对象原型的函数. 返回一个 Boolean， 如果更新成功，则返回true。
- Reflect.isExtensible(target)
  - 类似于 Object.isExtensible()
- Reflect.preventExtensions(target)
  - 类似于 Object.preventExtensions()。返回一个Boolean。
- Reflect.getOwnPropertyDescriptor(target, propertyKey)
  - 类似于 Object.getOwnPropertyDescriptor()。如果对象中存在该属性，则返回对应的属性描述符, 否则返回 undefined.
- Reflect.defineProperty(target, propertyKey, attributes)
  - 和 Object.defineProperty() 类似。如果设置成功就会返回 true
- Reflect.ownKeys(target)
  - 返回一个包含所有自身属性（不包含继承属性）的数组。(类似于 Object.keys(), 但不会受enumerable影响).
- Reflect.has(target, propertyKey)
  - 判断一个对象是否存在某个属性，和 in 运算符 的功能完全相同。
- Reflect.get(target, propertyKey[, receiver])
  - 获取对象身上某个属性的值，类似于 target[name]。
- Reflect.set(target, propertyKey, value[, receiver])
  - 将值分配给属性的函数。返回一个Boolean，如果更新成功，则返回true。
- Reflect.deleteProperty(target, propertyKey)
  - 作为函数的delete操作符，相当于执行 delete target[name]。
- Reflect.apply(target, thisArgument, argumentsList)
  - 对一个函数进行调用操作，同时可以传入一个数组作为调用参数。和 Function.prototype.apply() 功能类似。
- Reflect.construct(target, argumentsList[, newTarget])
  - 对构造函数进行 new 操作，相当于执行 new target(...args)。

Reflect的使用

- 那么我们可以将之前Proxy案例中对原对象的操作，都修改为Reflect来操作：

```
const objProxy = new Proxy(obj, {
  has: function(target, key) {
    return Reflect.has(target, key)
  },
  set: function(target, key, value) {
    return Reflect.set(target, key, value)
  },
  get: function(target, key) {
    return Reflect.get(target, key)
  },
  deleteProperty: function(target, key) {
    return Reflect.deleteProperty(target, key)
  }
})
```

Receiver的作用

- 我们发现在使用getter、setter的时候有一个receiver的参数，它的作用是什么呢？
  - 如果我们的源对象（obj）有setter、getter的访问器属性，那么可以通过receiver来改变里面的this；
- 我们来看这样的一个对象：

```
const objProxy = new Proxy(obj, {
  has: function(target, key) {
    return Reflect.has(target, key)
  },
  set: function(target, key, value, receiver) {
    console.log("set捕获器", key)
    return Reflect.set(target, key, value, receiver)
  },
  get: function(target, key, receiver) {
    console.log("get捕获器", key)
    return Reflect.get(target, key, receiver)
  },
  deleteProperty: function(target, key) {
    return Reflect.deleteProperty(target, key)
  }
})
```

Reflect的construct

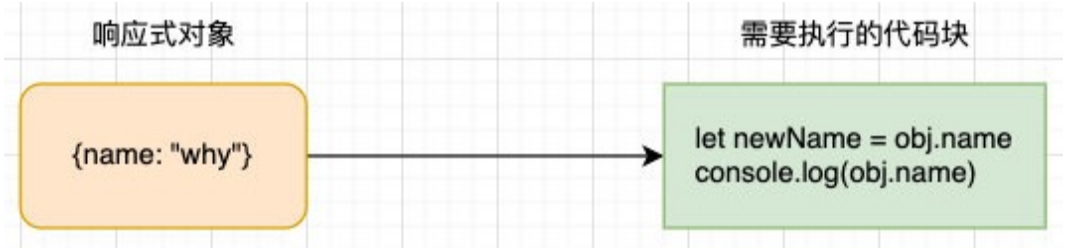
```
function Student(name, age) {
  this.name = name
  this.age = age
}

function Animal() {

}

const stu = Reflect.construct(Student, ["why", 18], Animal)
console.log(stu.__proto__ === Animal.prototype) // true
```

什么是响应式？

- 我们先来看一下响应式意味着什么？我们来看一段代码：
  - m有一个初始化的值，有一段代码使用了这个值；
  - 那么在m有一个新的值时，这段代码可以自动重新执行；

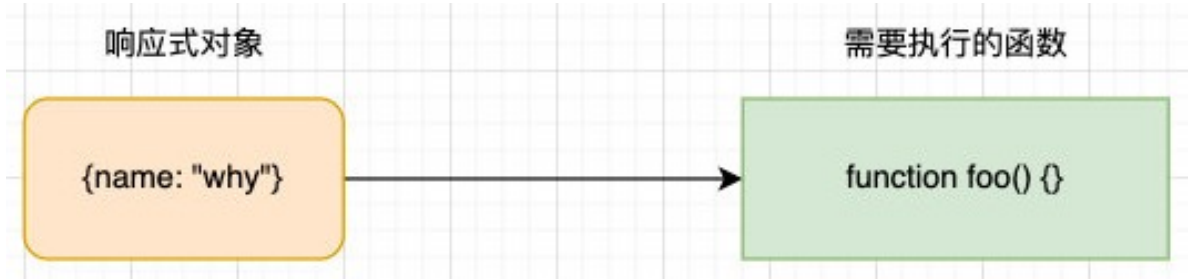
```
let m = 20
console.log(m)
console.log(m * 2)

m = 40
```

- 上面的这样一种可以自动响应数据变量的代码机制，我们就称之为是响应式的。
  - 那么我们再来看一下对象的响应式：



响应式函数设计

- 首先，执行的代码中可能不止一行代码，所以我们可以将这些代码放到一个函数中：
  - 那么我们的问题就变成了，当数据发生变化时，自动去执行某一个函数；



- 但是有一个问题：在开发中我们是有很多的函数的，我们如何区分一个函数需要响应式，还是不需要响应式呢？
  - 很明显，下面的函数中 foo 需要在obj的name发生变化时，重新执行，做出相应；
  - bar函数是一个完全独立于obj的函数，它不需要执行任何响应式的操作；

```
function foo() {
  let newName = obj.name
  console.log(obj.name)
}
```

```
function bar() {
  const result = 20 + 30
  console.log(result)
  console.log("Hello World")
}
```

响应式函数的实现watchFn

- 但是我们怎么区分呢？
  - 这个时候我们封装一个新的函数watchFn；
  - 凡是传入到watchFn的函数，就是需要响应式的；
  - 其他默认定义的函数都是不需要响应式的；

```
const reactiveFns = []

function watchFn(fn) {
  reactiveFns.push(fn)
  fn()
}
```

```
watchFn(function() {
  let newName = obj.name
  console.log(obj.name)
})

watchFn(function() {
  console.log("my name is " + obj.name)
})
```

响应式依赖的收集

- 目前我们收集的依赖是放到一个数组中来保存的，但是这里会存在数据管理的问题：
  - 我们在实际开发中需要监听很多对象的响应式；
  - 这些对象需要监听的不只是一个属性，它们很多属性的变化，都会有对应的响应式函数；
  - 我们不可能在全局维护一大堆的数组来保存这些响应函数；
- 所以我们要设计一个类，这个类用于管理某一个对象的某一个属性的所有响应式函数：
  - 相当于替代了原来的简单 reactiveFns 的数组；

```
class Depend {
  constructor() {
    this.reactiveFns = []
  }
  addDepend(fn) {
    this.reactiveFns.push(fn)
  }
  notify() {
    this.reactiveFns.forEach(fn => {
      fn()
    })
  }
}
```

```
const dep = new Depend()

function watchFn(fn) {
  dep.addDepend(fn)
  fn()
}
```

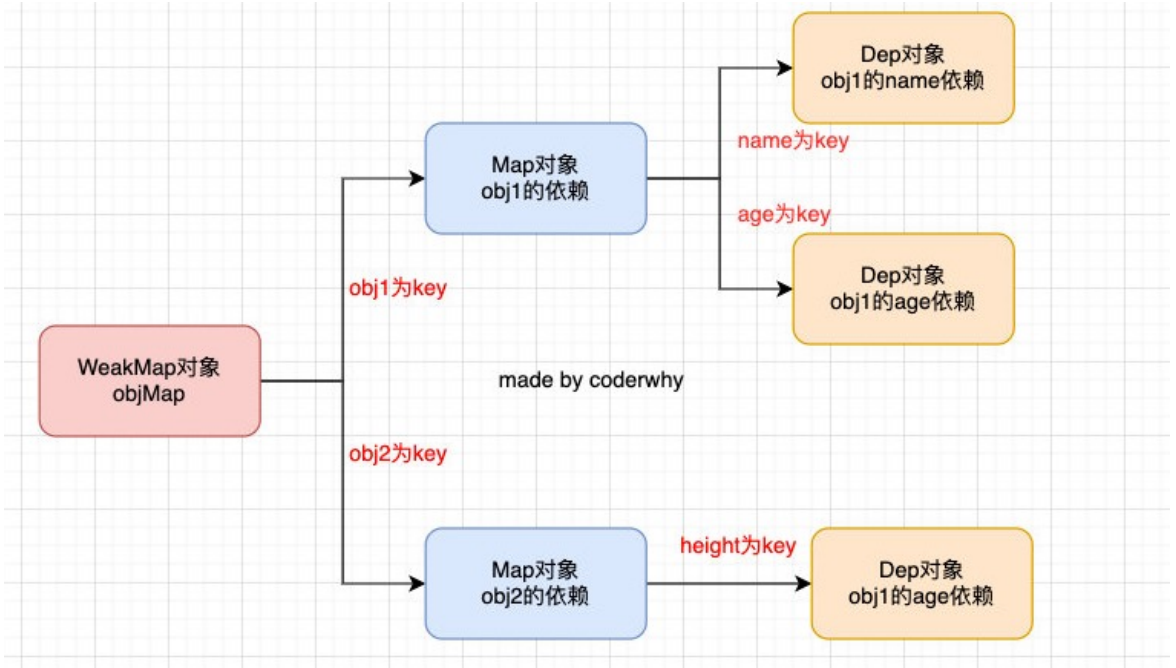
监听对象的变化

- 那么我们接下来就可以通过之前学习的方式来监听对象的变量：
  - 方式一：通过 Object.defineProperty的方式（vue2采用的方式）；
  - 方式二：通过new Proxy的方式（vue3采用的方式）；
- 我们这里先以Proxy的方式来监听：

```
const proxyObj = new Proxy(obj, {
  get: function(target, key, receiver) {
    Reflect.get(target, key, receiver)
  },
  set: function(target, key, value, receiver) {
    console.log("设置了新的值", key, value)
    Reflect.set(target, key, value, receiver)
    dep.notify()
  }
})
```

对象的依赖管理

- 我们目前是创建了一个Depend对象，用来管理对于name变化需要监听的响应函数：
  - 但是实际开发中我们会有不同的对象，另外会有不同的属性需要管理；
  - 我们如何可以使用一种数据结构来管理不同对象的不同依赖关系呢？
- 在前面我们刚刚学习过WeakMap，并且在学习WeakMap的时候我讲到了后面通过WeakMap如何管理这种响应式的数据依赖：



对象依赖管理的实现

- 我们可以写一个getDepend函数专门来管理这种依赖关系：

```
const targetMap = new WeakMap()
function getDepends(obj, key) {
  // 根据对象获取对应的Map对象
  let objMap = targetMap.get(obj)
  if (!objMap) {
    objMap = new Map()
    targetMap.set(obj, objMap)
  }

  // 根据key获取Depend对象
  let depend = objMap.get(key)
  if (!depend) {
    depend = new Depend()
    objMap.set(key, depend)
  }
  return depend
}
```

```
const proxyObj = new Proxy(obj, {
  get: function(target, key, receiver) {
    return Reflect.get(target, key, receiver)
  },
  set: function(target, key, value, receiver) {
    Reflect.set(target, key, value, receiver)
    const dep = getDepends(target, key)
    dep.notify()
  }
})
```

正确的依赖收集

- 我们之前收集依赖的地方是在 watchFn 中：
  - 但是这种收集依赖的方式我们根本不知道是哪一个key的哪一个depend需要收集依赖；
  - 你只能针对一个单独的depend对象来添加你的依赖对象；
- 那么正确的应该是在哪里收集呢？应该在我们调用了Proxy的get捕获器时
  - 因为如果一个函数中使用了某个对象的key，那么它应该被收集依赖；

```
let reactiveFn = null
function watchFn(fn) {
  // dep.addDepend(fn)
  reactiveFn = fn
  fn()
  reactiveFn = null
}
```

```
const proxyObj = new Proxy(obj, {
  get: function(target, key, receiver) {
    const dep = getDepends(target, key)
    dep.addDepend(reactiveFn)
    return Reflect.get(target, key, receiver)
  },
  set: function(target, key, value, receiver) {
    Reflect.set(target, key, value, receiver)
    const dep = getDepends(target, key)
    dep.notify()
  }
})
```

对Depend重构

- 但是这里有两个问题：
  - 问题一：如果函数中有用到两次key，比如name，那么这个函数会被收集两次；
  - 问题二：我们并不希望将添加reactiveFn放到get中，以为它是属于Dep的行为；
- 所以我们需要对Depend类进行重构：
  - 解决问题一的方法：不使用数组，而是使用Set；
  - 解决问题二的方法：添加一个新的方法，用于收集依赖；

```
class Depend {
  constructor() {
    this.reactiveFns = new Set()
  }
  addDepend(fn) {…
  }
  depend() {
    if (reactiveFn) {
      this.reactiveFns.add(reactiveFn)
    }
  }
  notify() {…
  }
}
```

```
const proxyObj = new Proxy(obj, {
  get: function(target, key, receiver) {
    const dep = getDepends(target, key)
    dep.depend()
    return Reflect.get(target, key, receiver)
  },
  set: function(target, key, value, receiver) {
    Reflect.set(target, key, value, receiver)
    const dep = getDepends(target, key)
    dep.notify()
  }
})
```

创建响应式对象

- 我们目前的响应式是针对于obj一个对象的，我们可以创建出来一个函数，针对所有的对象都可以变成响应式对象：

```
function reactive(obj) {
  return new Proxy(obj, {
    get: function(target, key, receiver) {
      const dep = getDepends(target, key)
      dep.depend()
      return Reflect.get(target, key, receiver)
    },
    set: function(target, key, value, receiver) {
      Reflect.set(target, key, value, receiver)
      const dep = getDepends(target, key)
      dep.notify()
    }
  })
}
```

```
const obj2 = reactive({
  address: "广州市"
})

watchFn(function() {
  console.log("我的地址:", obj2.address)
})

obj2.address = "北京市"
```

Vue2响应式原理

- 我们前面所实现的响应式的代码，其实就是Vue3中的响应式原理：
  - Vue3主要是通过Proxy来监听数据的变化以及收集相关的依赖的；
  - Vue2中通过我们前面学习过的Object.defineProerty的方式来实现对象属性的监听；
- 我们可以将reactive函数进行如下的重构：
  - 在传入对象时，我们可以遍历所有的key，并且通过属性存储描述符来监听属性的获取和修改；
  - 在setter和getter方法中的逻辑和前面的Proxy是一致的；

```
function reactive2(obj) {
  Object.keys(obj).forEach(key => {
    let value = obj[key]
    Object.defineProperty(obj, key, {
      get: function() {
        const dep = getDepends(obj, key)
        dep.depend()
        return value
      },
      set: function(newValue) {
        const dep = getDepends(obj, key)
        value = newValue
        dep.notify()
      }
    })
  })
  return obj
}
```